

什么是 Tkinter

本系列教程译自 [Fredrik Lundh](#) 的 [An Introduction to Tkinter](#)。

Tkinter（也叫 Tk 接口）是 Tk 图形用户界面工具包标准的 Python 接口。Tk 是一个轻量级的跨平台图形用户界面（GUI）开发工具。Tk 和 Tkinter 可以运行在大多数的 Unix 平台、Windows、和 Macintosh 系统。

Tkinter 由一定数量的模块组成。Tkinter 位于一个名为 `_tkinter`（较早的版本名为 `tkinter`）的二进制模块中。Tkinter 包含了对 Tk 的低级接口模块，低级接口并不会

被应用级程序员直接使用，通常是一个共享库（或 DLL），但是在一些情况下它也被 Python 解释器静态链接。除了 Tk 接口模块，Tkinter 也包含了一定数量的 Python 模块。其中两个最重要的模块是 Tkinter 本身和名为 Tkconstants 的模块。前者自动引导后者，因此使用 Tkinter，你首先需要做的是导入 Tkinter 模块，代码如下：

```
import Tkinter
```

或

```
from Tkinter import *
```

第一个 Tkinter 程序

一、代码如下：

#File: hello1.py

*from Tkinter import **

root = Tk()

w = Label(root, text="Hello, world!")

w.pack()

root.mainloop()

运行结果如下：



关闭这个窗口即可终止这个程序的运行。

二、对代码的说明：

我们通过导入 Tkinter 模块开始。Tkinter 模块包含了用 Tk 工具包工作所需的所有的类，函数和其它一些必须的东西。在大多数情况下，你只需要简单的从 Tkinter 导入所有的东西到你的模块的名字空间，如下所示：

```
from Tkinter import *
```

然后初始化 Tkinter,方法是我们必须先创建一个 Tk root

（根）窗口部件，它是一个普通的窗口，带有标题条和其它由你的窗口管理器供给的附属。你 应该只创建一个 root 窗口部件，这个 root 窗口部件必须在其它窗口部件创建之前创建。初始化 Tkinter 的代码如下：

```
root = Tk()
```

接着我们创建一个 Label（标签）窗口部件作为这个 root 窗口的孩子，代码如下：

```
w = Label(root, text="Hello, world!")
```

Label 窗口部件可以显示文本、图标或图象。我们这里用 text 选项来指定要显示的文本。

接下来我们在 Label 窗口部件上调用了 pack 方法，它告诉 Label 窗口部件去调整自己的尺寸来适应所给定文本

的大小，并且使用自己可见，代码如下：

```
w.pack()
```

但是在这些发生之前，我们必须进入 Tkinter 的事件循环，代码如下：

```
root.mainloop()
```

这个程序将一直处在事件循环之中，直到我们关闭了这个窗口。事件循环不仅仅处理来自用户的事件（如鼠标敲击和按键按下）或者窗口系统（重绘事件和窗口配置消息），它也处理来自 Tkinter 自身的工作等待队列，这些工作之一就包括由 pack 方法所产生的工作和显示更新。这就意味着这个应用程序窗口在进入这个事件循环之前将不会显示出来。

第二个 Tkinter 程序

当我们在写一个较大的程序的时候，把代码封装在类中通常是一个好的主意。

一、代码如下：

#File: hello2.py

*from Tkinter import **

class App:

def __init__(self, master):

frame = Frame(master)

frame.pack()

*self.button = Button(frame, text="QUIT",
fg="red", command=frame.quit)*

self.button.pack(side=LEFT)

self.hi_there = Button(frame, text="Hello",


```
command=self.say_hi)  
self.hi_there.pack(side=LEFT)
```

```
def say_hi(self):  
print "hi there, everyone!"
```

```
root = Tk()
```

```
app = App(root)
```

```
root.mainloop()
```

运行结果如下：



如果你点击 Hello 按钮，将在控制台打印出"hi there, everyone!"。如果你点击 QUIT 按钮，程序将终止。

二、代码说明：

这个简单的应用程序被写成了一个类。这个构造器（`__init__` 方法）通过一个父部件被调用，并针对父部件增加了一些子部件。构造器通过创建一个 Frame（帧）窗口部件开始。一个帧是一个简单的容器，在这个例子中，我们仅用来容纳另外的两个部件。

class App:

```
def __init__(self, master):
```

```
    frame = Frame(master)
```

```
    frame.pack()
```

这个帧实例被存储在一个名为 **Frame** 的局部变量中。在创建了这个部件后，我们立即调用 **pack** 方法来使用这个帧可见。

然后我们创建两个 **Button**（按钮）窗口部件作为这个帧的孩子。

```
self.button = Button(frame, text="QUIT", fg="red",
```

```
    command=frame.quit)
```

```
self.button.pack(side=LEFT)
```

```
self.hi_there = Button(frame, text="Hello",  
                        command=self.say_hi)  
self.hi_there.pack(side=LEFT)
```

这次，我们传递了一定数量的选项给构造器。第一个按钮被标为"QUIT"，字为红色（fg 是 foreground<前景色>的缩写）。第二个被标为"Hello"。两个按钮都有一个 command 选项。这个选项指定了一个函数或方法，在按钮被点击时会被调用。

按钮实例被存储在实例属性组中。side=LEFT 参数表示这两个按钮在帧中将被分开放置；第一个按钮被放置在帧的左边缘，第二个被放在第一个的右边（帧的左边缘仍保留着空格）。默认情况下，部件的放置都是相对于它们的父亲（frame 部件相对于 master，button 相对于

frame)。如果 side 选项没指定，side 默认值为 TOP。

"Hello"按钮的回调函数如下所示，它在按钮每次被按下时简单地打印一条信息给控制台：

```
def say_hi(self):  
    print "hi there, everyone!"
```

最后我们提供了一些脚本级的代码来创建一个 Tk root 部件，和一个 App 类的实例（这个实例使用这个 root 部件作为它的父亲）：

```
root = Tk()  
  
app = App(root)
```

root.mainloop()

`root.mainloop()`调用 `root` 部件的 `mainloop` 方法。它进入 Tk 事件循环，这个应用程序将处于这个 Tk 事件循环之中直到 `quit` 方法被调用（点击 QUIT 按钮），或窗口被关闭。

关于窗口部件引用

在第二个例子中，`frame` 部件被存储在一个名为 `frame` 的局部变量中，而 `button` 部件则存储在两个实例的属性组中。这是否隐藏着一个严重的问题呢：当 `__init__` 函数

返回并且 `frame` 变量超出了范围会发生什么呢？。

不要紧；这儿确实没有必要去保持对窗口部件实例的引用。Tkinter 自动维护一个窗口部件树（通过对象实例的 `master` 和 `children` 属性），因此，当应用程序的最后一个引用消失时窗口部件不会消失；窗口部件必须显示的用 `destroy` 方法销毁。但是如果你希望在窗口部件被创建以后用它来做一些事情，你最好保持对你自己的窗口部件实例的引用。

注意如果你不需要保持对一个窗口部件的引用，你可以用单独的一行来创建和 `pack`（包装）它，如下：

```
Button(frame, text="Hello",  
command=self.hello).pack(side=LEFT)
```

不要存储这个操作的结果，当你试图去用这个结果时你会很失望（因为 pack 方法返回 None）。为小心起见，最好将将 pack（包装）分开，如下所示：

```
w = Button(frame, text="Hello", command=self.hello)  
w.pack(side=LEFT)
```

关于窗口部件的名字

另一个容量引起混淆的方面（尤其是使用 Tcl 编程 Tk 的有一些经验的人）是 Tinter 的窗口部件的名字的概念。

在 Tcl 中，你必须显示的命名每一个窗口部件。例如下面的 Tcl 命令创建一个名为 ok 的按钮作为名为 dialog 的窗口部件的孩子（dialog 又是 root 窗口的孩子）：

button.dialog.ok

相应的 Tkinter 调用将如下：

ok = Button(dialog)

在这个 Tkinter 案例中，ok 和 dialog 是对窗口部件实例的引用，不是窗口部件实际的名字。因为 Tk 自身需要这些名字，Tkinter 自动为每个新窗口部件赋一个唯一的名字。在这个 Tkinter 案例中，这个 dialog 的名字或许类似 ".1428748"，并且 button 可能是 ".1428748.1432920"。如果你希望得到一个 Tkinter 窗口部件的完整名字，你可以在这个窗口部件实例上使用 str 函数：

```
>>> print str(ok)  
.1428748.1432920
```

如果你确实需要为一个窗口部件指定一个名字，你可以在你创建这个窗口部件时使用 **name** 选项。你这么做的原因可能是你需要和用 Tcl 写的代码接口。

下面的例子将产生一个名为".dialog.ok"的窗口部件（如果你忘了命名 **dialog**,那么名字可能类似".1428748.ok"）：

```
ok = Button(dialog, name="ok")
```

为了避免与此同时 Tkinter 的名字机制相冲突，不要使用只包含数字的名字。同样注意 **name** 是只能创建一次的选

项；一旦你创建了这个部件的名字，那么你就不能再改变它的名字了。

Tkinter 类之窗口部件类

Tkinter 支持 15 个核心的窗口部件，这个 15 个核心窗口部件类列表如下：

窗口部件及说明：

Button:

一个简单的按钮，用来执行一个命令或别的操作。

Canvas:

组织图形。这个部件可以用来绘制图表和图，创建图形编辑器，实现定制窗口部件。

Checkbutton:

代表一个变量，它有两个不同的值。点击这个按钮将会在这两个值间切换。

Entry:

文本输入域。

Frame:

一个容器窗口部件。帧可以有边框和背景，当创建一个应用程序或 `dialog`(对话) 版面时，帧被用来组织其它的窗口部件。

Label:

显示一个文本或图象。

Listbox:

显示供选方案的一个列表。`listbox` 能够被配置来得到 `radiobutton` 或 `checklist` 的行为。

Menu:

菜单条。用来实现下拉和弹出式菜单。

Menubutton:

菜单按钮。用来实现下拉式菜单。

Message:

显示一文本。类似 label 窗口部件，但是能够自动地调整文本到给定的宽度或比率。

Radiobutton:

代表一个变量，它可以有多个值中的一个。点击它将为这个变量设置值，并且清除与这同一变量相关的其它 radiobutton。

Scale:

允许你通过滑块来设置一数字值。

Scrollbar:

为配合使用 `canvas`, `entry`, `listbox`, and `text` 窗口部件的标准滚动条。

Text:

格式化文本显示。允许你用不同的样式和属性来显示和编辑文本。同时支持内嵌图象和窗口。

Toplevel:

一个容器窗口部件，作为一个单独的、最上面的窗口显示。

注意在 Tkinter 中窗口部件类没有分级；所有的窗口部件类在树中都是兄弟。

所有这些窗口部件提供了 Misc 和几何管理方法、配置管理方法和部件自己定义的另外的方法。此外，Toplevel 类也提供窗口管理接口。这意味一个典型的窗口部件类提供了大约 150 种方法。

Button 窗口部件

Button（按钮）窗口部件是一个标准的 Tkinter 窗口部件，

用来实现各种按钮。按钮能够包含文本或图象，并且你能够将按钮与一个 Python 函数或方法相关联。当这个按钮被按下时，Tkinter 自动调用相关联的函数或方法。

按钮仅能显示一种字体，但是这个文本可以跨行。另外，这个文本中的一个字母可以有下划线，例如标明一个快捷键。默认情况，Tab 键用于将焦点移动到一个按钮部件。

一、那么什么时候用按钮部件呢？

简而言之，按钮部件用来让用户说“马上给我执行这个任务”，通常我们用显示在按钮上的文本或图象来提示。按钮通常用在工具条中或应用程序窗口中，并且用来接收

或忽略输入在对话框中的数据。

关于按钮和输入的数据的配合，可以参看 `Checkbutton` 和 `Radiobutton` 部件。

二、样式

普通的按钮很容易被创建，仅仅指定按钮的内容（文本、位图、图象）和一个当按钮被按下时的回调函数即可：

```
b = Button(master, text="OK", command=self.ok)
```

没有回调函数的按钮是没有用的，当你按下这个按钮时它什么也不做。你可能在开发一个应用程序的时候想实现这种按钮，比如为了不干扰你的 beta 版的测试者：

```
b = Button(master, text="Help", state=DISABLED)
```

如 果你没有指定尺寸，按钮的大小将正好能够容纳它的内容。你可以用 `padx` 和 `pady` 选项来增加内容与按钮边框的间距。你也可以用 `height` 和 `width` 选项来显式地设置按钮的尺寸。如果你在按钮中显示文本，那么这些选项将以文本的单位为定义按钮的尺寸。如果你替而代之显示图象，那么按钮的尺寸将是象素（或其它的屏幕单位）。你实际上甚至能够用象素单位来定义文本按钮的尺寸，但这可能带来意外的结果。下面是指定尺寸的一段例子代码：

```
f = Frame(master, height=32, width=32)  
f.pack_propagate(0) # don't shrink  
b = Button(f, text="Sure!")
```

b.pack(fill=BOTH, expand=1)

按钮能够显示多行文本（但只能用一种字体）。你可以使用多行或 `wrplength` 选项来使按钮自己调整文本。当调整文本时，使用 `anchor, justify`, 也可加上 `padx` 选项来得到你所希望的格式。一个例子如下：

b = Button(master, text=longtext, anchor=W, justify=LEFT, padx=2)

为了使一个普通的按钮看起来像凹入的，例如你想去实现某种类型的工具框，你可简单地将 `relief` 的值从 "RAISED" 改变为 "SUNKEN"：

b.config(relief=SUNKEN)

你也可能想改变背景。注意：一个大概更好的解决方案是使用一个 Checkbutton 或 Radiobutton 其 indicatoron 选项的值设置为 false：

```
b = Checkbutton(master, image=bold, variable=var,  
indicatoron=0)
```

三、方法

Button 窗口部件支持标准的 Tkinter 窗口部件接口，加上下面的方法：

flash()：频繁重画按钮，使其在活动和普通样式下切换。

`invoke()`：调用与按钮相关联的命令。

下面的方法与你实现自己的按钮绑定有关：

`tkButtonDown()`, `tkButtonEnter()`, `tkButtonInvoke()`,
`tkButtonLeave()`, `tkButtonUp()`

这些方法可以用在定制事件绑定中，所有这些方法接收 0 个或多个形参。

四、选项

Button 窗口部件支持下面的选项：

`activebackground`, `activeforeground`

类型：颜色；

说明：当按钮被激活时所使用的颜色。

anchor

类型：常量；

说明：控制按钮上内容的位置。使用 N, NE, E, SE, S, SW, W, NW, or CENTER 这些值之一。默认值是 CENTER。

background (bg), foreground (fg)

类型：颜色；

说明：按钮的颜色。默认值与特定平台相关。

bitmap

类型：位图；

说明：显示在窗口部件中的位图。如果 image 选项被指定了，则这个选项被忽略。下面的位图在所有平台上都有效：error, gray75, gray50, gray25, gray12, hourglass, info, questhead, question, 和 warning.



这后面附加的位图仅在 Macintosh 上有效：document, stationery, edition, application, accessory, folder, pfolder, trash, floppy, ramdisk, cdrom, preferences, querydoc, stop, note, 和 caution.

你也可以从一个 XBM 文件中装载位图。只需要在 XBM 文件名前加一个前缀@,例如"@sample.xbm".

borderwidth (bd)

类型：整数；

说明：按钮边框的宽度。默认值与特定平台相关。但通常是 1 或 2 像素。

command

类型：回调；

说明：当按钮被按下时所调用的一个函数或方法。所回调的可以是一个函数、方法或别的可调用的 Python 对象。

cursor

类型：光标；

说明：当鼠标移动到按钮上时所显示的光标。

default

类型：常量；

说明：如果设置了，则按钮为默认按钮。注意这个语法在 Tk 8.0b2 中已改变。

disabledforeground

类型：颜色；

说明：当按钮无效时的颜色。

font

类型：字体；

说明：按钮所使用的字体。按钮只能包含一种字体的文本。

highlightbackground, highlightcolor

类型：颜色；

说明：控制焦点所在的高亮边框的颜色。当窗口部件获得焦点的时候，边框为 `highlightcolor` 所指定的颜色。否则边框为 `highlightbackground` 所指定的颜色。默认值由系统所定。

highlightthickness

类型：距离；

说明：控制焦点所在的高亮边框的宽度。默认值通常是 1 或 2 象素。

image

类型： 图象；

说明： 在部件中显示的图象。如果指定，则 text 和 bitmap 选项将被忽略。

justify

类型： 常量；

说明： 定义多行文本如何对齐。可取值有： LEFT, RIGHT, 或 CENTER。

padx, pady

类型： 距离；

说明： 指定文本或图象与按钮边框的间距。

relief

类型：常量；

说明：边框的装饰。通常按钮按下时是凹陷的，否则凸起。另外的可能取值有 GROOVE, RIDGE, 和 FLAT。

state

类型：常量；

说明：按钮的状态：NORMAL, ACTIVE 或 DISABLED。
默认值为 NORMAL。

takefocus

类型：标志；

说明：表明用户可以 Tab 键来将焦点移到这个按钮上。
默认值是一个空字符串，意思是如果按钮有按键绑定的话，它可以通过所绑定的按键来获得焦点。

text

类型：字符串；

说明：显示在按钮中的文本。文本可以是多行。如果 bitmaps 或 image 选项被使用，则 text 选项被忽略。

textvariable

类型：变量；

说明：与按钮相关的 Tk 变量（通常是一个字符串变量）。如果这个变量的值改变，那么按钮上的文本相应更新。

underline

类型：整数；

说明：在文本标签中哪个字符加下划线。默认值为-1，

意思是没有字符加下划线。

width, height

类型：距离；

说明：按钮的尺寸。如果按钮显示文本，尺寸使用文本的单位。如果按钮显示图象，尺寸以像素为单位（或屏幕的单位）。如果尺寸没指定，它将根据按钮的内容来计算。

wraplength

类型：距离；

说明：确定一个按钮的文本何时调整为多行。它以屏幕的单位为单位。默认不调整。

Mixins

Tkinter 模块提供了相应于 Tk 中的各种窗口部件类型的类和一定数量的 mixin 和别的帮助类（mixin 是一个类，被设计来使用多态继承与其它的类结合）。当你使用 Tkinter 时，你不将直接访问 mixin 类。

一、实施 mixins

通过 root 窗口和窗口部件类，Misc 类被用作 mixin。它提供了大量的 Tk 和窗口相关服务，这些服务对所有 Tkinter 核心窗口部件者有效。这些通过委托完成；窗口

部件仅仅直接请求适当的内部对象。

Wm 类通过 root 窗口和顶级窗口部件类被用作 mixin。通过委托它提供了窗口管理服务。

使用委托像这样简化你的应用程序代码：一旦你有一窗口部件，你能够使用这个窗口部件的实例的方法访问 Tkinter 的所有部份。

二、Geometry(几何学)与 mixins

Grid, Pack, Place 这些类通过窗口部件类被用作 mixins。通过委托，它们也提供了访问不同几何管理的支持。

下面是 Geometry Mixins 的列表及说明：

管理器及说明：

Grid: `grid` 几何管理器允许你通过在一个二维网格中组织窗口部件来创建一个类似表的版面。

Pack: `pack` 几何管理器通过在一个帧中把窗口部件包装到一个父部件中来创建一个版面。为了对窗口部件使用这个几何管理器，我们在这个窗口部件上使用 `pack` 方法来集成。

Place: `place` 几何管理器让你显式将一个窗口部件放到给定的位置。要使用这个几何管理器，需使用 `place` 方法。

三、窗口部件配置管理

`Widget` 类使用 `geometry mixins` 来混合 `Misc` 类，并通过

cget 和 configure 方法来增加配置管理，也可以通过一个局部的字典接口。

窗口部件的配置

要配置一个窗口部件的外观，你用选项比使用方法调用好。典型的选项包括 text、color、size、command 等等。对于处理选项，所有的核心窗口部件执行同样的配置接口：

配置接口

widgetclass(master, option=value, ...) => widget

说明:

创建这个窗口部件的一个实例，这个实例作为给定的 master 的孩子，并且使用给定的选项。所有的选项都有默认值，因此在简单的情况下，你仅需要指定这个 master。如果你想的话，你也可以不指定 master；Tkinter 这时会使用最近创建的 root 窗口作为 master。注意这个 name 选项仅能在窗口部件被创建时设置。

cget(option) => string

说明:

返回一个选项的当前值。选项的名字和返回值都是字符串。要得到 name 选项，使用 str(widget)代替。

configure(option=value, ...), config(option=value, ...)

说明:

设置一个或多个选项（作为关键字参数给定）。

注意一些选项的名字与 Python 中的保留字相同 (class, from 等)。要使用这些作为关键字参数，仅需要在这些选项名后添加一下划线(class_, from_)。注意你不能
用此方法来设置 name 选项；name 选项只能在窗口部件
被创建时设置。

为了方便起见，窗口部件也实现一个局部的字典接口。
__setitem__ 方法映射 configure，而 __getitem__ 方法映射
cget。你可以使用下面的语法来设置和查询选项：

value = widget[option]

widget[option] = value

注意每个赋值都导致一个对 Tk 的调用。如果你希望去改变多个选项，单独地调用(`config` 或 `configure`)去改变它们是一个好的主意。

这下面的字典方法也适用于窗口部件：

keys() => *list*

说明：

返回窗口部件中所有可以被设置的选项的一个列表。
`name` 选项不包括在这个列表中（它不能通过字典接口被查询或修改）。

向后兼容性

关键字参数在 Python1.3时被引入。之前，使用原始的 Python 字典将选项传递给窗口构造器和 `configure` 方法。原代码类似如下：

```
self.button = Button(frame, {"text": "QUIT", "fg": "red",  
                             "command": frame.quit})  
self.button.pack({"side": LEFT})
```

关键字参数语法更优雅和少容易发生错误。但是为了与存在的代码兼容，Tkinter 仍支持老的语法。在新的程序中你不应用再用老的语法，即使是在某些情况下是很有吸引力的。例如，如果你创建了一个定制的窗口部件，它需要沿它的父类传递配置选项，你的代码可能如下：

```
def __init__(self, master, **kw):
```

Canvas.__init__(self, master, kw) # kw 是一个字典
上面的代码在当前版本的 Tkinter 下工作的很好，但是它在将来的版本下可能不工作。一个通常的办法是使用

apply 函数：

```
def __init__(self, master, **kw):
```

```
    apply(Canvas.__init__, (self, master), kw)
```

这个 apply 函数使用了一个函数（一个未约束的方法），一个带参数的元组（它必须包括 self，因为我们调用一个未约束的方法），一个可选的，提供了关键字参数的字典。

窗口部件的样式之颜色

所有的 Tkinter 标准窗口部件提供了一套样式设置选项，这让你可以去修改这些窗口部件的外观如颜色、字体和其它的可视外观。

颜色

大部份窗口部件都允许你指定窗口部件和文本的颜色，这可以使用 `background` 和 `foreground` 选项。要指定颜色，

你可以使用颜色名，也可以使用红、绿、蓝颜色组合。

1、颜色名

Tkinter 包括一个颜色数据库，它将颜色名映射到相应的 RGB 值。这个数据库包括了通常的名称如 Red, Green, Blue, Yellow, 和 LightBlue，也可使用外来的如 Moccasin, PeachPuff 等等。在 X window 系统上，颜色名由 X server 定义。你能够找到一个名为 xrgb.txt 的文件，它包含了一个由颜色名和相应 RGB 值组成的列表。在 Windows 和 Macintosh 系统上，颜色名表内建于 Tk 中。

在 Windows 下，你可以使用 Windows 系统颜色（用户可以通过控制面板来改变这些颜色）：

*SystemActiveBorder, SystemActiveCaption,
SystemAppWorkspace, SystemBackground,
SystemButtonFace, SystemButtonHighlight,
SystemButtonShadow, SystemButtonText,
SystemCaptionText, SystemDisabledText, SystemHighlight,
SystemHighlightText,
SystemInactiveBorder, SystemInactiveCaption,
SystemInactiveCaptionText, SystemMenu,
SystemMenuText, SystemScrollbar, SystemWindow,
SystemWindowFrame, SystemWindowText。*

在 Macintosh 上，下面的系统颜色是有效的：

*SystemButtonFace, SystemButtonFrame, SystemButtonText,
SystemHighlight, SystemHighlightText, SystemMenu,*

*SystemMenuActive, SystemMenuActiveText,
SystemMenuDisabled, SystemMenuText,
SystemWindowBody。*

颜色名是大小写不敏感的。许多颜色名词与词之间有无格都有效。例如"lightblue", "light blue", 和 "Light Blue"都是同一颜色。

2、RGB 格式

如果你需要显式地指定颜色名，你可以使用如下格式的字符串：

#RRGGBB

RR, GG, BB 分别是 red, green 和 blue 值的十六进制表示。

下面的例子演示了如何将一个颜色三元组转换为

一个 Tk 颜色格式：

```
tk_rgb = "#%02x%02x%02x" % (128, 192, 200)
```

Tk 也支持用形如"#RGB"和"rrrrggggbbbb"去分别指定16和65536程度之间的值。

你可以使用窗口部件的 `winfo_rgb` 方法来将一个代表颜色的字符串（名字或 RGB 格式）转换为一个三元组：

```
rgb = widget.winfo_rgb("red")
```

```
red, green, blue = rgb[0]/256, rgb[1]/256, rgb[2]/256
```

注意 `winfo_rgb` 返回16位的 RGB 值，范围在0~65535之间。要将它们映射到更通用的0~255范围内，你必须将每个值

都除以256（或将它们向右移8位）。

窗口部件的样式之字体

字体

窗口部件允许你显示文本和指定所使用的字体。所有的窗口部件都提供了合理的默认值，你很少需要去为简单元素如标签和按钮指定字体。

字体通常使用 `font` 窗口部件选项指定。Tkinter 支持一定数量的不同字体描述类型：

* Font descriptors

* User-defined font names

* System fonts

* X font descriptors

Tk8.0以前的版本仅 X font 描述被支持。

1、字体描述

从 Tk8.0开始，Tkinter 支持独立于平台的字体描述。你可以使用元组来指定一个字体，这个元组包含了一个字体类型名字，一个以磅为单位的高度，代表一个或多个

样式的字符串。例如：

("Times", 10, "bold")

("Helvetica", 10, "bold italic")

("Symbol", 8)

要得到默认的尺寸和类型，你可以给出作为单一字符串的字体名。如果这个字体类型名字没有包括空格，你也可以给这个字符串自身增加尺寸和样式：

"Times 10 bold"

"Helvetica 10 bold italic"

"Symbol 8"

在大部份 Windows 平台上存在如下有效的字体类名：
Arial (相 应 于 Helvetica), Courier New (Courier), Comic

Sans MS, Fixedsys, MS Sans Serif, MS Serif, Symbol,
System, Times New Roman (Times), 和 Verdana:

arial 14 points: I'd like to have an argu

courier new 12 points: What? Pri

comic sans ms 8 points: Pack my box with fiven dozen jugs of

fixedsys 9 points: Here you see some Eng

ms sans serif 11 points: Pack my box with fiven dozen

ms serif 16 points: The quick brown fox ju

σψμβολ 12 πουντα: Φιγυρς τηισ ουτ, ωονδερ βοψ

system 10 points: Hello, Harry. Now there's the s

times new roman 16 points: That turni

verdana 10 points: The quick brown fox jumps c

注意：如果这个字体类型名包含空格，你必须使用上面所描述的元组语法。

有效的样式有 `normal`, `bold`, `roman`, `italic`, `underline`, and `overstrike`。

Tk8.0自动映射 `Courier`, `Helvetica`, 和 `Times` 到所有平台上相应的本地字体类型名。此外，在 Tk8.0下字体格式不会引起问题，如果 Tk 不能找出确切的匹配，它会试着找类似的字体，如果失败，Tk 就使用特定平台的默认字体。

Tk4.2在 Windows 下同样支持这种字体描述。这儿有几个限制，包括字体类型名必须在平台上存在，并非这所有

上面样式名都存在（或它们中的一些有不同的名字）。

2、字体名

此外，Tk8.0允许你去创建已命名的字体并且当为一个窗口部件指定字体时使用它们的名字。

tkFont 模块提供一个 Font 类，这个类允许你去创建字体实例。你可以随处使用这样一个实例。你也可能使用一个字体实例来得到字体的量度，包括存在于那个字体中的字符串所站用的尺寸。

```
tkFont.Font(family="Times", size=10,  
             weight=tkFont.BOLD)  
tkFont.Font(family="Helvetica", size=10,
```

```
weight=tkFont.BOLD,  
    slant=tkFont.ITALIC)  
tkFont.Font(family="Symbol", size=8)
```

如果你修改一个已命名的字体（使用 `config` 方法），这个改变将自动影响到所有使用这个字体的窗口部件。

`Font` 构造器支持下列的样式选项（注意常量被定义在 `tkFont` 模块中）：

样式选项及说明：

family 选项

类型：字符串

说明：字体类型

size 选项

类型：整型

说明：以磅为单位的字体的尺寸。要以像素为单位的话，使用负值。

weight 选项

类型：常量

说明：字体的粗细。使用 NORMAL 或 BOLD。默认为 NORMAL。

slant 选项

类型：常量

说明：字体倾斜。使用 NORMAL 或 ITALIC。默认为 NORMAL。

underline 选项

类型：标志

说明：字体下划线。如果1(true)，字体加下划线。默认为0(false)。

overstrike 选项

类型：标志

说明：字体划线。如果为1(true)，则字体上有一条线；默认为0(false)。

3、系统字体

Tk 也支持特定系统的字体名。在 X 下，这些通常是字体别名如 fixed,6x10 等等。

在 Windows 下，这些包括
ansi,ansifixed,device,oemfixed,system 和 systemfixed:

```
ansi: I didn't know ants had six legs, Marcus
ansifixed: Another merciless sweep
device: We like dressing up, ye
oemfixed: One day Ricky the magic p
system: Pretty strong meat there from Sam
systemfixed: Simon Zinc Trumpet Har
```

在 Macintosh 上，系统字体名是 application 和 system。

注意：系统字体是字体名，不是字体类型名，它们不能与尺寸或样式属性结合。为了可移植性，尽可能避免使用这些名字。

4、X 字体描述

X 字体描述是如下格式的字符串（星号所代表的是无关字段。具体细节可查看 Tk 文档或 X 手册）：

`-*-family-weight-slant-*--*-size-*-*-*-*-charset`

典型的字体类别如：Times, Helvetica, Courier or Symbol.

weight 可以是"Bold"或"Normal"。slant 取值中 R 代表"roman"(正常), I 代表"italic", o 代表团"oblique" (实际上等同于 italic)。

size 是字体的高度, 以十分之一磅为单位。一英寸72磅, 但是一些低分辨率的显示器的1磅较常规的大些, 以便小字体能够清晰显示。charset (字符集) 通常是 ISO8859-1 (ISO Latin 1), 但一些字体也使用其它的值。

下面的描述的 family 取值是 Times, weight 取值是 Bold, slant 取值是 R, size 取值是 120, charset 取值是 ISO8859-1:

-*-Times-Bold-R-*--*-120-*-*-*-*ISO8859-1

如果你不关心 `charset` (字符集), 或你使用如 `Symbol` 的字体 (这种字体类别有特定的字符集), 那么你可以使用一个星号作为描述的最后部分:

`-*-Symbol-*-*-*-80-*`

典型的 X server 至少支持 `Times`, `Helvetica`, `Courier` 等字体, `size` 有 8, 10, 12, 14, 18, 和 24 磅, `weight` 有 `normal`, `bold`, `italic`(`Times`)或 `oblique`(`Helvetica`, `Courier`)。大多数的服务器都有支持随意查看字体。你可以使用如 `xlsfonts` 和 `xfontsel` 来检查你所访问的服务器的字体。

这种类型的字体描述可以用在 `Windows` 和 `Macintosh` 上。注意: 如果你使用 `Tk4.2`, 你必须牢记字体类型必须是 `Windows` 所支持的一种。

格式化文本

虽然文本标签和按钮通常包含单行文本，但 Tkinter 也支持多行。要分离文本到多行，只需要在必要的地方插入换行符（\n）。

默认情况下，文本居中。你也可以通过设置 `justify` 选项为 `LEFT` 或 `RIGHT` 来改变文本的位置。默认值是 `CENTER`。

你也可以使用 `wraplength` 选项来设置一个最大宽度，并且

让窗口部件自己调整多行文本。如果窗口部件太窄，它将使单个词跨行。

边框

所有的 Tkinter 窗口部件都有边框（尽管对于某些窗口部件默认是不显示边框的）。边框由一个可选的3D 浮雕和一个焦点高亮区域组成。

一、relief

这个 relief 设置控制如何绘制窗口部件的3D 边框：

`borderwidth`(或 `bd`)是边框的宽度，以像素为单位。大多数的窗口部件都有一个默认的1或2像素的边框宽度。

`relief` 的取值可以有如下几种：

SUNKEN, RAISED, GROOVE, RIDGE, and FLAT.

二、焦点高亮

这个高亮设置控制如何指示窗口部件（或它的孩子之一）获得了按键焦点。在大多数情况下，这个高亮区域是在 `relief` 外面的边框。下面的选项控制如何绘制另外的边框：

highlightcolor 用来绘制当窗口部件获得按键焦点的高亮

区域。通常是黑色或别的明显的对比色。

highlightbackground 用来绘制当窗口部件没获得焦点的高亮区域。通常与窗口部件的背景色一样。

highlightthickness 选项是高亮区域的宽度，以像素为单位。对于那些获得按钮焦点的窗口部件通常是1或2个像素

三、光标

这个 `cursor` 选项控制当鼠标移动到窗口部件上时使用哪种鼠标光标。如果这个选项没有设置，这个窗口部件将使用和它父亲一样的鼠标指针。

注意：一些窗口部件（包括 `Text` 和 `Entry` 窗口部件）默认设置 `cursor` 选项。

下面是一些内建的鼠标光标的样式：



事件和绑定

正如早些时候，一个 Tkinter 应用程序大部分时间花费在事件循环中（通过 `mainloop` 方法进入事件循环）。事件来自不同的消息，包括用户按下按键和鼠标操作，和来自于窗口管理器的重绘事件（在许多情况下不是由用户直接引起）。

Tkinter 提供了强大的机制让你可以自己处理事件。对于任一窗口部件，你可以为事件绑定 Python 函数和方法：

```
widget.bind(event, handler)
```


如果发生在窗口部件中的事件与所描述的事件匹配，那么所给定的 `handler`(处理器)将被描述这个事件的对象调用。

下面是一个捕获窗口中的点击的例子：

#File: bind1.py

*from Tkinter import **

root = Tk()

def callback(event):

print "clicked at", event.x, event.y

```
frame = Frame(root, width=100, height=100)
frame.bind("<Button-1>", callback)
frame.pack()

root.mainloop()
```

上面这个例子，我们使用了 frame 窗口部件的 bind 方法为一个事件调用<Button-1>绑定了一个回调函数。运行这个程序并在所显示的窗口中点击。每次点击，在控制台窗口中都会打印一条如"clicked at 44 63"这样的信息。

一、事件

事件以字符串的形式给出，使用特定的语法：

<modifier-type-detail>

type 字段是事件区分符中最重要的部分。它指定了我们希望去绑定的事件种类，如用户的按钮、按键动作，或如 Enter, Configure 等窗口管理器事件。modifier 和 detail 字段被用来给出附加的信息，在多数情况下可以不用。这儿也有不同的方法去简化事件字符串；例如，要匹配一个键盘键，你可以不用尖括号而只使用这个键。当然，除非它的一个空格或一个尖括号。

下面我们给出最常用的事件及其说明：

<Button-1>事件

说明：鼠标按钮在窗口部件上按下。button 1 是鼠标左

按钮，`button 2` 是鼠标中间的按钮（如果有的话），`button 3` 是鼠标右按钮。当你在 一窗口部件上按下一个鼠标按钮时，Tkinter 将自动抓取鼠标指针，在鼠标按钮被按下时鼠标事件将被发送给当前窗口部件。鼠标指针相对于窗口部件的当前位置被提供给传递给回调函数的 `event` 对象的 `x` 和 `y` 成员中。

你可以用 `ButtonPress` 代替 `Button`，或只用 `<1>`（这等同于 `<Button-1>`, `<ButtonPress-1>`）。

<B1-Motion>事件

说明：鼠标移动并且鼠标左键被按住（`B2` 代表鼠标中间按键，`B3` 代表鼠标右按键）。鼠标指针相对于窗口部件的当前位置被提供给传递给回调函数的 `event` 对象的 `x` 和 `y` 成员中。

<ButtonRelease-1>事件

说明：鼠标左键释放。鼠标指针相对于窗口部件的当前位置被提供给传递给回调函数的 `event` 对象的 `x` 和 `y` 成员中。

<Double-Button-1>事件

说明：鼠标左键双击。你可以用 `Triple` 代替 `Double`。

<Enter>事件

说明：鼠标指针位于窗口部件中（这个事件不代表回车）

<Leave>事件

说明：鼠标指针离开窗口部件。

<Return>事件

说明：用户按下 Enter 键。你可以绑定到键盘上的所有实际上的键。对于通常的 102 键盘，这专用键有 Cancel, BackSpace, Tab, Enter, Shift_L, Control_L, Alt_L, Pause, Caps_Lock, Escape, PageUp, Page Down, End, Home, Left, Up, Right, Down, Print, Insert, Delete, F1, F2, F3, F4, F5, F6, F7, F8, F9, F10, F11, F12, Num_Lock, 和 Scroll_Lock。

<Key>事件

说明：用户按下任一键。这个键被提供给传递给回调函数的 event 对象的 char 成员中。

a 事件

说明：用户键入"a"。大部分的可打印字符都可替换 a。这个例外是空格（<space>） 和小于（<less>）。

<Shift-Up> 事件

说明：用户按下向上箭头，同时按住 shift 键。你可以用 Alt，Control 代替 shift。

<Configure> 事件

说明：窗口部件的尺寸改变（某些平台上是位置改变）。新的尺寸被提供给传递给回调函数的 event 对象的 width 和 height 属性中。

二、Event 对象

Event 对象是一个标准的 Python 对象实例，它的属性及说明如下：

widget

说明：产生事件的窗口部件。这是一个有效的 Tkinter 窗口部件的实例，不是一个名字。这个属性为所有事件设置。

x,y

说明：当前鼠标指针的位置，以像素为单位。

x_root,y_root

说明：当前鼠标相对于屏幕左上方的位置，以像素为单位。

char

说明：字符代码（仅键盘事件），是一个字符串。

keysym

说明：键的符号（仅键盘事件）。

keycode

说明：键的代码（仅键盘事件）。

num

说明：鼠标按钮的数字（仅鼠标按钮事件）。

`width,height`

说明：窗口部件的新尺寸，以像素为单位（仅 `Configure` 事件）

`type`

说明：事件类型。

为了兼容性，你应坚持使用 `char`, `height`, `width`, `x`, `y`, `x_root`, `y_root`, 和 `widget` 除非你确切地知道你在做什么。

三、实例和类绑定

上面的例子中我们使用 `bind` 方法创建一个实例绑定。这

意味着这个绑定仅应用于单个窗口部件；如果你创建新的 frame,它们将不继承这个绑定。

但是 Tkinter 也允许你在类和应用程序层上创建绑定；事实上，你可以在下面四个不同的层上创建绑定：

- *窗口部件实例，使用 `bind` 方法。

- *窗口部件的顶层窗口（顶层或 `root`），也使用 `bind` 方法。

- *窗口部件类，使用 `bind_class` 方法（这是 Tkinter 提供的标准绑定）。

- *整个应用程序，使用 `bind_all` 方法。

例如，你可以使用 `bind_all` 来为 F1 键创建一个到应用程序的绑定，这样你就可以在这个应用程序的任何地方提

供帮助。但是如果你为同一个键创建多个绑定或提供重叠的绑定会发生什么呢？

首先，对于上面四种层别之一的任何一种，Tkinter 选择有效绑定的最近匹配。例如，如果你为<Key>和<Return>事件创建了实例绑定，那么如果你按下了 Enter 键则只是这第二个绑定将被调用。

然而，如果你增加一个<Return>绑定到顶层窗口部件，则两个绑定都会被调用。Tkinter 首先调用实例层的绑定，然后调用顶层窗口层绑定，然后是类层（通常是一个标准绑定），最后是应用程序层。因此在极端的情况下，一个事件可以调用四个事件处理器。

一个引起混淆的情况是什么时候你使用绑定去覆盖一个标准窗口部件的默认行为。例如，假设你希望 Enter 在文本窗口部件中无效，以使用户不能在文本中插入新行。下面的代码可以实现吗？

```
def ignore(event):  
    pass  
text.bind("<Return>", ignore)
```

或用下面一行代码：

```
text.bind("<Return>", lambda e: None)
```

不幸的是，这个新行仍然插入了，因为上面的绑定仅应用于实例层，页标准的行为是由类层的绑定提供。

你可以使用 `bind_class` 方法来修改类层上的绑定，但是这

将改变应用程序中的所有文本窗口部件的行为。

一个简单的解决方法是阻止 Tkinter 传递事件给别的处理器；并且你的事件处理器只返回"break"字符串：

```
def ignore(event):  
    return "break"  
text.bind("<Return>", ignore)
```

或

```
text.bind("<Return>", lambda e: "break")
```

顺便说说，如果你真想改变应用程序中的所有文本窗口部件的行为，可以使用 `bind_class` 方法：

```
top.bind_class("Text", "<Return>", lambda e: None)
```

但是基于很多原因你是不会这样做的。例如，如果你希望使用从网上下载的一些酷而小的 UI 组件来扩展你的应用程序，这样做将会将应用程序完全搞乱。较好的方法是使用你自己定制的文本窗口部件类，这样就可以保持 Tkinter 默认绑定的完好：

```
class MyText(Text):  
    def __init__(self, master, **kw):  
        apply(Text.__init__, (self, master), kw)  
        self.bind("<Return>", lambda e: "break")
```

四、协议(protocols)

除了事件绑定，Tkinter 也支持一个称作协议处理器的机制。在这里，协议这个术语是指应用程序与窗口管理器

之间的交互。最常被使用的协议名为 `WM_Delete_WINDOW`，它被用来定义当用户显式使用窗口管理器关闭一个窗口时发生什么。

你可以使用 `protocol` 方法去为这个协议安装一个处理器（窗口部件必须是 `root` 或顶层窗口部件）：

```
widget.protocol("WM_Delete_WINDOW", handler)
```

一旦你安装了你自己的处理器，`Tkinter` 将不再自动关闭窗口。作为代替，比如你可以显示一个信息框询问用户当前数据是否保存。要从处理器关闭窗口，你只需要调用这个窗口的 `destroy` 方法：

例子、捕获 `destroy` 事件：

```
#File: protocol1.py
```



```
from Tkinter import *  
import tkMessageBox
```

```
def callback():  
    if tkMessageBox.askokcancel("Quit", "Do you really  
wish to quit?"):  
        root.destroy()
```

```
root = Tk()  
root.protocol("WM_Delete_WINDOW", callback)  
  
root.mainloop()
```

注意：即使你在顶层窗口上不为 WM_Delete_WINDOW 注册一个处理器，这个窗口自己也会照常被销毁。然而，对于 Python 1.5.2, Tkinter 将不会销毁相应的窗口部件实例层，因此你自己注册一个处理器是一个好的主意：

```
top = Toplevel(...)
```

```
# 确使用窗口部件实例被删除
```

```
top.protocol("WM_Delete_WINDOW", top.destroy)
```

五、其它的协议

窗口管理器协议是 X window 系统最初的部分（它们被定义在标题为 Inter- Client Communication Conventions Manual 文档，或 ICCCM 中）。在那个平台上你也可以安

装别的协议，如 WM_TAKE_FOCUS 和 WM_SAVE_YOURSELF。详细请阅 ICCCM 文档。

应用程序窗口

一、基本的窗口

先前我们使用了一个简单的例子，它仅在屏幕上显示一个窗口——root 窗口。这是当我们调用 Tk 构造器时自动创建的，对于简单的应用程序非常方便：

```
from Tkinter import *
```

```
root = Tk()
```

这里可以创建窗口内容作为 *root* 窗口的孩子

```
root.mainloop()
```

如果你要创建额外的窗口，你可以使用 `Toplevel` 窗口部件。它仅在屏幕上创建一个新的窗口，它的样式和行为都很象原本的 `root` 窗口：

```
from Tkinter import *
```

```
root = Tk()
```

创建 *root* 窗口内容

top = Toplevel()

创建顶层窗口内容

root.mainloop()

这里不必要使用 `pack` 去显示这个 `Toplevel`，因为它是自动被窗口管理器显示（事实上，如果你试图对 `Toplevel` 窗口部件使用 `pack` 或别的 `geometry` 管理器，你将会得到一个错误信息）。

二、菜单

Tkinter 为菜单提供了专门的窗口部件类型。要创建一个菜单，你要创建一个 Menu 类的实例，并且用 add 方法为这个实例增加条目：

- * `add_command(label=string, command=callback)` 增加一个普通的菜单条目。

- * `add_separator()` 增加一个分隔线。用来分组菜单条目。

- * `add_cascade(label=string, menu=menu instance)` 增加一个子菜单。这是一个依附于父亲的下拉菜单或折叠式菜单。

下面这个例子创建了一个小的菜单：

#File: menu1.py

```
from Tkinter import *
```

```
def callback():  
print "called the callback!"
```

```
root = Tk()
```

```
#create a menu
```

```
menu = Menu(root)
```

```
root.config(menu=menu)
```

```
filemenu = Menu(menu)
```

```
menu.add_cascade(label="File", menu=filemenu)
```

```
filemenu.add_command(label="New", command=callback)
    filemenu.add_command(label="Open...",
        command=callback)
    filemenu.add_separator()
filemenu.add_command(label="Exit", command=callback)

helpmenu = Menu(menu)
menu.add_cascade(label="Help", menu=helpmenu)
    helpmenu.add_command(label="About...",
        command=callback)

mainloop()
```

在上面的例子中，我们通过创建一个 **Menu** 实例开始，

然后我们使用 `config` 方法将这个菜单实例隶属于 `root` 窗口。这个菜单的内容将被用来去创建一个位于 `root` 窗口顶部的菜单条。你不必 `pack` 这个菜单，因为它被 Tkinter 自动显示。

接下来，我们创建一个新的 `Menu` 实例，使用菜单条作为这个新实例的父亲，并且使用 `add_cascade` 方法使用它成为一个下拉式菜单。然后我们调用 `add_command` 来给这个菜单添加命令（注意在这个例子中所有的命令都使用相同的回调函数），并使用 `add_separator` 在 `file` 命令和 `exit` 命令之间添加一个分隔线。

最后我们以同样的方式创建一个小的帮助菜单。

三、工具条

许多应用程序都在菜单条下面放置了一个工具条，通常都包含了一定量的按钮以完成通常的功能如 open,file,print,undo 等等。

下面的例子，我们使用一个 Frame 部件作为一个工具条，并放置上一定量的通常的按钮：

```
# File: toolbar1.py
```

```
from Tkinter import *
```

```
root = Tk()
```

```
def callback():  
    print "called the callback!"
```

```
# create a toolbar  
toolbar = Frame(root)
```

```
b = Button(toolbar, text="new", width=6,  
            command=callback)  
b.pack(side=LEFT, padx=2, pady=2)
```

```
b = Button(toolbar, text="open", width=6,  
            command=callback)  
b.pack(side=LEFT, padx=2, pady=2)
```

toolbar.pack(side=TOP,fill=X)

mainloop()

按钮以左边对齐，工具条自身放置在最上面（将 fill 设置为 X）。

注意：这里我使用文本标签，而没有使用图标，这样更简单。要显示一个图标，你可以用 **PhotoImage** 构造器来从磁盘装载一个小的图象，并使用 **image** 选项去显示它。

四、状态条

大多数应用程序都在应用程序窗口的底部显示一个状态

条。用 Tkinter 实现一个状态条是很容易的：你可以简单地使用一个合适的可配置的 Label 窗口部件，而后重新配置这个 text 选项。这里有一个方法：

```
status = Label(master, text="", bd=1, relief=SUNKEN,  
                anchor=W)  
status.pack(side=BOTTOM, fill=X)
```

如果你希望有想象力，你可以使用下面的类来代替。它在类中封装了一个 label 窗口部件，并提供了 set 和 clear 方法来修改内容：

```
# File: tkSimpleStatusBar.py  
  
class StatusBar(Frame):
```

```
def __init__(self, master):  
    Frame.__init__(self, master)  
    self.label = Label(self, bd=1, relief=SUNKEN,  
                        anchor=W)  
    self.label.pack(fill=X)
```

```
def set(self, format, *args):  
    self.label.config(text=format % args)  
    self.label.update_idletasks()
```

```
def clear(self):  
    self.label.config(text="")  
    self.label.update_idletasks()
```

这个 `set` 方法类似于 C 的 `printf` 函数；它要求一个格式字符串，后或许跟一套参数（缺点是如果你希望打印任意的字符串，你必须这样做 `set("%s", string)`）。同样注意这个方法调用 `update_idletasks` 方法以保证绘制操作（如状态条更新）立即进行。你可以使用通常的窗口部件语法创建并显示这个状态条：

```
status = StatusBar(root)  
status.pack(side=BOTTOM, fill=X)
```

标准对话框之消息框

以前我们关注了在应用程序工作区域中放置的一些窗口

部件，现在让我们来关注 GUI 程序设计的另一重要部分：
显示对话框和消息框。

从 Tk4.2开始，Tk 库提供了一套标准对话框，它们可以被用来显示消息框，和选择文件与颜色。另外，Tkinter 提供了一些简单的对话框以让你可以去要求用户的整数、浮点数值和字符串。这些标准对话框尽可能使用特定平台的机制以得到相应的外观。

一、消息框

tkMessageBox 模块提供了一个对消息框的接口。

使用这个模块最容易的方法是使用这些便利的函数之

一：showinfo, showwarning, showerror, askquestion, askokcancel, askyesno, 或 askretrycancel。它们有同样的语法：

tkMessageBox.function(title, message [, options])。

这个 title 参数显示在窗口的标题中，这个 message 显示在对话框体中。你可以在消息中使用换行符("\n")以使消息占据多行。options 可以被用来修改外观。

这第一组的标准对话框被用来呈现信息。你提供标题和消息，这些函数使用一个恰当的图标显示这些，当用户按下 OK 时返回。返回值被忽略。

这有个例子：

try:

```
fp = open(filename)
except:
tkMessageBox.showwarning(
    "Open file",
    "Cannot open this file\n(%s)" % filename
)
return
```

下面是第一组标准对话框(showinfo, showwarning, showerror dialogs)的图示：



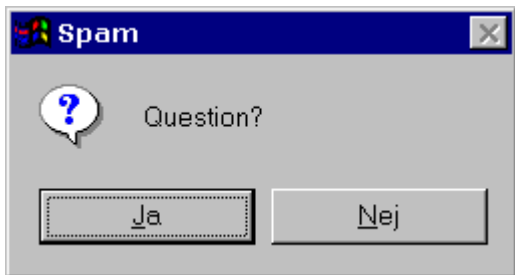


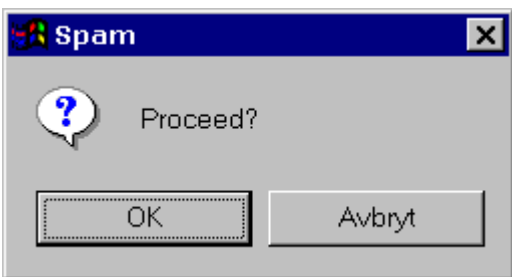


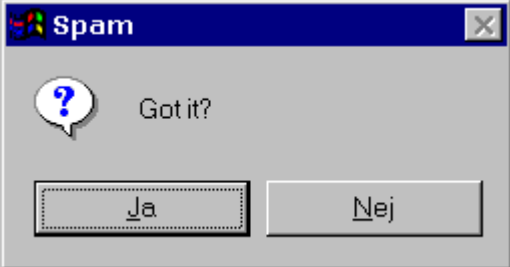
这第二组被用来询问问题。`askquestion` 函数返回字符串 "yes"或"no"（你可以使用 `options` 去修改所显示的按钮的数量和类型），例如：

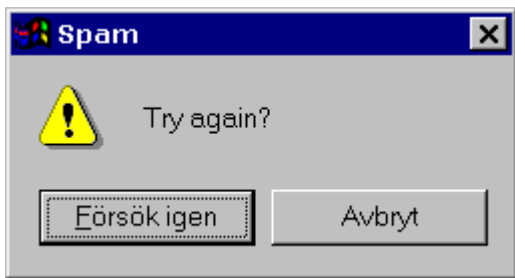
```
if tkMessageBox.askyesno("Print", "Print this  
report?"):  
    report.print()
```

第二组(askquestion, askokcancel, askyesno, askretrycancel dialogs)的图示:









二、消息框选项

如果标准的消息框不适合你的要求，你可以使用 options 来达到你的要求。

下面是消息框选项及说明：

default

类型：常量

说明：默认键：ABORT, RETRY, IGNORE, OK, CANCEL, YES 或 NO(它们定义在 tkMessageBox 模块中)。

icon

类型：常量

说明：要显示的图标：ERROR, INFO, QUESTION, 或 WARNING。

message

类型：字符串

说明：要显示的消息(函数的第二个参数)。

parent

类型：窗口部件

说明：在哪个窗口的顶部放置消息框。当消息框关闭后，焦点将返回父窗口。

title

类型：字符串

说明：消息框标题(函数的第一个参数)。

type

类型：常量

说明：消息框类型；即显示哪个按钮：

ABORTRETRYIGNORE, OK, OKCANCEL,
RETRYCANCEL, YESNO, 或 YESNOCANCEL。

数据输入

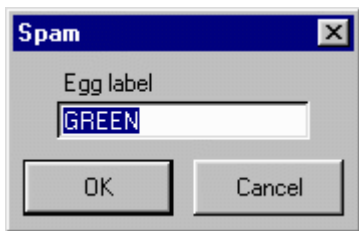
tkSimpleDialog 模块提供了一个针对如下简单对话框的接口。

一、字符串

tkSimpleDialog 中的 askstring 函数用于给用户一个提示字符串。你指定这个对话框的标题和提示字符串，当用户关闭对话框时函数返回。提示字符串可以包含换行符：

tkSimpleDialog.askinteger(title, prompt [,options])。要求用户输入一个字符串值。如果用户按下了 Enter 键或敲击了 OK，那么函数返回这个字符串。如果用户通过按下 Esc 或敲击 Cancel 或显式地由窗口管理器关闭了这个对话框，则这个函数返回 None。

askstring 的图示：



下面是这个函数能够使用的选项（options）及说明：

initialvalue

类型：字符串

说明：初始值。默认值是一个空字符串。

parent

类型：窗口部件

说明：对话框放置于哪个窗口上。当对话框关闭时，焦点将返回到父窗口。

二、数字值

`askinteger` 和 `askfloat` 函数类似于 `askstring` 函数，但它们只分别接收整型和浮点数值。你也可以使用 `minvalue` 和

maxvalue 选项来限制输入的范围：

tkSimpleDialog.askinteger(title, prompt [,options])。要求用户输入一个整形值。如果用户输入的值不是一个有效的整数或浮点数值，一个消息框将显示，并且这个对话框不会关闭。如同 askstring 函数，如果对话框被取消则函数返回 None。

tkSimpleDialog.askfloat(title, prompt [,options])。同样，函数返回一个浮点数值。

askinteger, askfloat 图示如下：

Spam ✕

Egg count

Spam ✕

Egg weight
(in tons)

这两个函数可以使用的选项及说明如下：

initialvalue

类型：整形或浮点数

说明：初始值。默认是一个空字符串。

parent

类型：窗口部件

说明：对话框放置于哪个窗口上。当对话框关闭时，焦点将返回到父窗口。

minvalue

类型：整形或浮点数

说明：最小值。如果低于，当用户敲击 OK 时一个消息

框将显示，并且对话框不会关闭。

maxvalue

类型：整形或浮点数

说明：最小值。如果高于，当用户敲击 OK 时一个消息框将显示，并且对话框不会关闭。

三、文件名

tkFileDialog 模块（包含在早先说明的标准对话框工具包中）可以用来从用户得到文件名。这个模块提供了两个方便的函数，一个用来得到已存在的文件名以便于你打开它，另一个用来得到一个文件名以保存东西。

tkFileDialog.askopenfilename([options])。如果这个对话框被取消，则函数返回 `None`。




tkFileDialog.asksaveasfilename([options])。

下面是 `askopenfilename`, `asksaveasfilename` 的图示：

Öppna

Leta i:  introduction



 dialog1.py
 dialog2.py
 dialog3.py

Filnamn:

Filformat:





















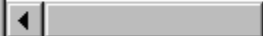
Spara som

Spara i:

 introduction



- | | | |
|---|--|---|
|  askokcancel.gif |  bind1.py |  dialog3.gif |
|  askopenfilename.gif |  button.htm |  dialog3.py |
|  askquestion.gif |  colorchooser.gif |  frame.htm |
|  askretrycancel.gif |  dialog1.py |  grid.htm |
|  askyesno.gif |  dialog2.gif |  grid1.gif |
|  bg3.gif |  dialog2.py |  grid1.py |



Filnamn:

Filformat:

All Files (*.*)

下面是 options 及说明：

`defaultextension`

类型：字符串

说明：文件名的后缀，如果用户没有显示指定。字符串应该包含开头的点号（打开对话框将忽略）

`filetypes`

类型：列表

说明：由（标签，匹配模式）元组构成的序列。同一个标签可以有几个匹配模式。使用*作为模式表明所有文件。

initialdir

类型：字符串

说明：初始目录。

initialfile

类型：字符串

说明：初始文件（打开对话框将忽略）。

parent

类型：窗口部件

说明：对话框放置于哪个窗口上。当对话框关闭时，焦点将返回到该窗口。

title

类型：字符串

说明：消息框标题。

四、颜色

`tkColorChooser` 模块（包含在早先说明的标准对话框工具包中）可以被用来指定一个 RGB 颜色值。

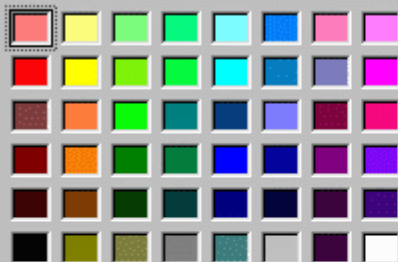
tkColorChooser.askcolor([color[,options]])。这个函数返回两个值；第一个是一个三元组（包含了代表红、绿、蓝三色的三个整数值(0~255)，第二个是 Tk 颜色字符串。当你显示这个对话框时要预定一个颜色，你可以传递一个颜色给这个函数。

如果对话框被取消，则函数返回（None,None）

下面是 askcolor 的图示：

Färg

Grundfärger:



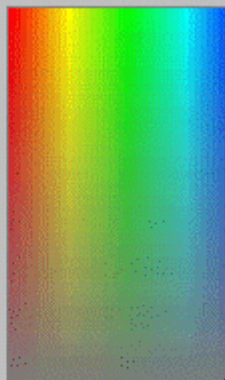
Egna färger:



Definiera egna färger>>

OK

Avbryt



Nyan

Mättna

Färg|Ren färg

Ljusstyrk

Lägg till eg

下面是 askcolor 的 Options 及说明：

initialcolor

类型：颜色

说明：当对话框显示时所标记的颜色（可以是 RGB 值或颜色名）

parent

类型：窗口部件

说明：对话框放置于哪个窗口上。当对话框关闭时，焦点将返回到该窗口。

title

类型：字符串

说明：消息框标题。

对话框

虽然前面部分所介绍的标准对话框对许多简单的应用是足够了，但是大多数较大的应用要求更复杂的对话框。例如，要为一个应用程序设置配置参数，你也许想在一个对话框中让用户输入更多的信息。

基本上，创建一个对话框与创建一个应用程序窗口没什么不同。仅仅是使用这个 `Toplevel` 部件，填充必要的

输入域，按钮和其他一些窗口部件。（顺便说一下，不要使用 `ApplicationWindow` 类，他只会让你的用户糊涂。）

下面的例子中，`MyDialog` 类创建了一个 `Toplevel` 窗口部件，并在其上增加了一些窗口部件。然后使用 `wait_window` 来等待直到对话框被关闭。如果用户敲击 OK，这个输入域中的值将被打印，并且对话框被明确地销毁。

```
#File: dialog1.py
```

```
from Tkinter import *
```

```
class MyDialog:
```

```
def __init__(self, parent):
```

```
    top = self.top = Toplevel(parent)
```

```
    Label(top, text="Value").pack()
```

```
    self.e = Entry(top)
```

```
    self.e.pack(padx=5)
```

```
    b = Button(top, text="OK", command=self.ok)
```

```
    b.pack(pady=5)
```

```
def ok(self):
```

print "value is", self.e.get()

self.top.destroy()

root = Tk()

Button(root, text="Hello!").pack()

root.update()

d = MyDialog(root)

root.wait_window(d.top)

注意：我们这儿没有调用 `mainloop` 方法；使用 `wait_window` 来进入局部事件循环就足够了。但是这个例子有一些问题：

* `root` 窗口仍旧是活动的。当对话框显示的时候你也可以敲击 `root` 窗口上的按钮。如果对话框依赖于当前应用程序状态，那么让用户自己干预应用程序可能是灾难性的。并且仅显示多个对话框无疑会使用你的用户糊涂。

* 你必须在输入域中敲击以使光标移入输入域中，同样必须敲击按钮。在输入域中按下 `Enter` 键是不够的。

* 这里应该有一些控制方法来取消对话框（作为我们早先学习的，我们也应该处理 `WM_Delete_WINDOW` 协议）。

针对第一个问题，Tkinter 提供了一个名为 `grab_set` 的方法，它确保没有鼠标或键盘事件被传送给错误的窗口。

第二个问题由几个部分组成；首先我们需要明确地移动键盘焦点到对话框。这可以使用 `focus_set` 方法做到。第二，我们需要绑定 Enter 键以使它调用 `ok` 方法。这个容易，只需在 `Toplevel` 窗口部件上使用 `bind` 方法（并且确保修改 `ok` 方法，我们给了它一个可选的参数以使它不对 `event` 对象阻塞）。

第三个问题可以通过增加一个额外的 Cancel 按钮它调用 `destroy` 方法，并且当用户按下了 Esc 或明确地关闭了窗口时使用 `bind` 和 `protocol` 来完成同样任务同样的事。

下面的 `Dialog` 类提供了上面所有这些，和一些额外的技巧。要实现你自己的对话框，你可以简单地继承这个类并覆盖 `body` 和 `apply` 方法。这个前部分创建对话框体，后部分在用户敲击 OK 时被调用。

#File: tkSimpleDialog.py

```
from Tkinter import *  
import os
```

```
class Dialog(Toplevel):
```

```
    def __init__(self, parent, title = None):
```

```
Toplevel.__init__(self, parent)  
self.transient(parent)
```

```
if title:  
    self.title(title)
```

```
self.parent = parent
```

```
self.result = None
```

```
body = Frame(self)  
self.initial_focus = self.body(body)  
body.pack(padx=5, pady=5)
```

self.buttonbox()

self.grab_set()

if not self.initial_focus:

self.initial_focus = self

self.protocol("WM_Delete_WINDOW",

self.cancel)

self.geometry("+%d+%d"

%

(parent.winfo_rootx()+50,

parent.winfo_rooty()+50))

self.initial_focus.focus_set()

self.wait_window(self)

#

construction hooks

def body(self, master):

create dialog body. return widget that should

have

initial focus. this method should be overridden

pass

def buttonbox(self):

add standard button box. override if you don't

want the

#standard buttons

box = Frame(self)

w = Button(box, text="OK", width=10,

command=self.ok, default=ACTIVE)

w.pack(side=LEFT, padx=5, pady=5)

w = Button(box, text="Cancel", width=10,

command=self.cancel)

w.pack(side=LEFT, padx=5, pady=5)

```
self.bind("<Return>", self.ok)  
self.bind("<Escape>", self.cancel)
```

```
box.pack()
```

```
#
```

```
# standard button semantics
```

```
def ok(self, event=None):
```

```
    if not self.validate():
```

```
        self.initial_focus.focus_set() # put focus back
```

```
        return
```

self.withdraw()

self.update_idletasks()

self.apply()

self.cancel()

def cancel(self, event=None):

#put focus back to the parent window

self.parent.focus_set()

self.destroy()


```
#  
# command hooks  
  
def validate(self):  
  
    return 1 # override  
  
def apply(self):  
  
    pass # override
```

主要的技巧在构造器中实现。首先，`transient` 将这个窗口与一个父窗口相关联(通常是引起这个对话框的应用程序窗口)。对话框不会以图标形式显示在窗口管理

器中（例如，在 Windows 下，对话框不会出现在任务栏中），如果你将父窗口图标化，那么对话框也将隐藏。接下来，构造器创建对话框，然后调用 `grab_set` 设置对话框模式，`geometry` 布置对话框相对于窗口的位置，`focus_set` 移动焦点到适当的窗口部件（通常是使用 `body` 方法返回的窗口部件），最后 `wait_window`。

注意我们使用 `protocol` 方法来确保一个明确的关闭被当作 `cancel`(取消)，并且在 `buttonbox` 方法中，我们绑定 `Enter` 键到 `OK`，`Esc` 到 `Cancel`。`default=ACTIVE` 调用标记这个 `OK` 按钮为特定平台的默认按钮方式。

使用这类比摸清它是如何实现的容易的多；只需要在 `body` 方法中创建必要的窗口部件，在 `apply` 方法中取得

结果和执行你想做的。

下面是一个简单的例子：

#File: dialog2.py

import tkSimpleDialog

class MyDialog(tkSimpleDialog.Dialog):

def body(self, master):

Label(master, text="First: ").grid(row=0)

Label(master, text="Second: ").grid(row=1)

self.e1 = Entry(master)

```
self.e2 = Entry(master)
```

```
self.e1.grid(row=0, column=1)
```

```
self.e2.grid(row=1, column=1)
```

```
return self.e1 #initial focus
```

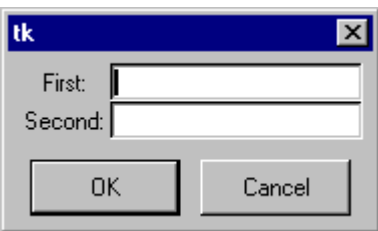
```
def apply(self):
```

```
    first = string.atoi(self.e1.get())
```

```
    second = string.atoi(self.e2.get())
```

```
    print first, second #or something
```

运行 dialog2.py 结果如下：



注意当对话框显示的时候，这个 `body` 方法可以随意地返回应该获得焦点的窗口部件。假如这与你的对话框无关，则只返回 `None`（或者省略 `return` 语句）。

上面的例子在 `apply` 方法中执行实际的处理。但是代替在 `apply` 方法中的处理，你可以存储所键入的数据到一个实例的属性中：

```
def apply(self):
```

```
first = int(self.e1.get())  
second = int(self.e2.get())  
self.result = first, second
```

```
d = MyDialog(root)  
print d.result
```

注意如果对话框被取消了，则 `apply` 方法不会被调用，并且 `result` 属性没有被设置。这个对话框构造器设置这个属性为 `None`，以便于在处理之前你可以简单地测试这个结果。如果你想返回数据到其它的属性中，请确保在 `body` 方法中初始化它们（或简单地在 `apply` 方法中设置 `result` 为 1，并在文章其它属性之前测试它）。

格子布置

我们在设计应用程序窗口时的所使用的方便的 pack 管理器在对话框的应用中确不是那么容易使用。一个典型的对话框可能包括一定数量的输入域和复选框，和相应的标签。考虑如下的简单例子：

例：简单对话框的布置

First:	<entry field>
Second:	<entry field>
<checkbutton>	

要用 pack 管理器来实现它，我们可以创建一个 frame 来包含标签"first:"，和相对应的输入域，并当 pack 它们时使用"side=LEFT"。为下一行增加一个相应的 frame，

并使用"side=TOP"pack这些 frames 和 checkbox 到一个外部的 frame。不幸的，使用 这个方式 pack 标签则可能得到输入域的列队，并且如果我们替换为使用"side=RIGHT"来 pack 输入域，如果输入域有不同的宽度则事情将破坏。通过仔细地使用 width、size 等等选项，我们可以努力得到合适的结果。但是这儿有一个非常容易的方法：使用 grid 管理器。

grid 管理器把主窗口部件(典型的是一个 frame)分成一个二维的格子或表。对于每个窗口部件，你只需要指定它出现在格子的哪儿，其余就由 grid 管理器去操心吧。下面的 body 方法显示了如何得到上面的布局：

例子：使用 grid geometry 管理器

#File: dialog3.py


```
def body(self, master):
```

```
Label(master, text="First: ").grid(row=0, sticky=W)
```

```
Label(master, text="Second: ").grid(row=1, sticky=W)
```

```
self.e1 = Entry(master)
```

```
self.e2 = Entry(master)
```

```
self.e1.grid(row=0, column=1)
```

```
self.e2.grid(row=1, column=1)
```

```
self.cb = Checkbutton(master, text="Hardcopy")
```

```
self.cb.grid(row=2, columnspan=2, sticky=W)
```

每个窗口部件将由 `grid` 管理器处理，你需要调用 `grid` 方法并使用 `row` 和 `column` 选项来告诉管理器把窗口部件放在哪儿。最上面一行和最左一列的数值是 0（这也是一个默认值）。这时 `checkboxbutton` 被放置在 `label` 和 `entry` 窗口部件的下面，`columnspan` 选项用来使它占据多个单元格。结果如下：



如果你看仔细点，你将发现这个对话框与 `dialog2.py` 的对话框有一点不同。这里，标签是左对齐的。如果你比较下代码，你将发现唯一的不同是一个名为 `sticky` 的选项。

当显示 `frame` 窗口部件时，`grid geometry` 管理器遍历所有的窗口部件，为每行每列计算出一个合适的宽度和高度。如果单元格大于窗口部件，则窗口部件是默认居中的。`sticky` 选项被用来修改这个行为。通过设置它为 `E`, `W`, `S`, `N`, `NW`, `NE`, `SE`, `SW` 之一，你能够与单元格的任一边或角对齐。但是如果有必要，你也可以使用这个选项去伸展窗口部件；如果你设置这个选项为 `E+W`，则这个窗口部件将伸展为占据单元格的整个宽度。如果设置为 `E+W+N+S`（或 `NW+SE` 等），窗口部件将两个方向都伸

展。实际上，这个 sticky 选项代替了通过 pack 管理器使用的 fill,expand,anchor 选项。

grid 管理器提供了许多其它的选项让你去调整结果布局的外观和行为。

确认数据

如果用户键入的数据不对怎么办？在我们当前的例子中，如果输入域中的内容不是一个整数 apply 方法将引发一个异常。我们当然可以用一个 try/except 和一个标准信

息框来处理这个：

```
def apply(self):  
    try:  
        first = int(self.e1.get())  
        second = int(self.e2.get())  
        dosomething((first, second))  
    except ValueError:  
        tkMessageBox.showwarning(  
            "Bad input",  
            "Illegal values, please try again"  
        )
```

这 里有一个问题：当 `apply` 方法被调用时，这个 `ok` 方法已经将对话框从屏幕上去除，并且我们一返回它就将其销毁。这个设计是有意的；如果我们在 `apply` 方法中执

行一些潜在冗长的处理，假如在我们完成之前对话框没被去除那将是非常混乱的。Dialog 类已经包含了钩子以实现另一方案：一个单独的 `validate` 方法，它在对话框被去除之前调用。

在下面的例子中，我们简单地将 `apply` 中的代码移到 `validate` 中，并且改变为存储结果到实例的属性中。

```
def validate(self):  
    try:  
        first = int(self.e1.get())  
        second = int(self.e2.get())  
        self.result = first, second  
        return 1  
    except ValueError:
```

```
tkMessageBox.showwarning(  
    "Bad input",  
    "Illegal values, please try again"  
)  
return 0
```

```
def apply(self):  
    dosomething(self.result)
```